IN THE SPECIFICATION

Please amend paragraphs [0026], [0028. [0033], [0034], [0040], [0042], and [0044], as follows:

[0026] The foreign thread code 101 may be designed with an expectation that the foreign thread code 101 is supported by a foreign platform 110. A portion of the foreign code 101 may [[used]] use other libraries 111 and thread libraries 112. Furthermore, foreign platform 110 may also include an operating system 113 and a foreign instruction set architecture (ISA) 114 that is understood by a foreign processor 115. An ISA is a specification of a set of all binary codes, also known as the opcodes or machine language that are commands in native forms to be understood by a particular computer processing unit (CPU). For example, the Intel.TM. 64-bit processor such as the Itanium Processor Family (IPF) and Extended Memory Technology (EM64T) processors may be designed with a particular ISA. The ISA supporting the Intel.TM. 64-bit processor may be referred to as ISA-64 and the ISA supporting the Intel.TM. 32-bit processor may be referred to as ISA-32.

[0028] After the foreign thread code 101 is transferred to the host platform 150, the foreign thread code 102 may have to rely on the other libraries 151 to ~~supported~~ support the specific library function calls as discussed above. In addition, the foreign thread code 102 may also rely on thread libraries 152 for proper thread initialization, creation, and termination. For example, the thread libraries 152 may be a foreign thread library that is capable of managing and supporting the foreign thread code 102.

[0033] FIG. 3 depicts the encapsulated components of the dynamic binary translator depicted in FIG. 2 according to an embodiment of the invention. In this ~~illustrating~~ illustration, a multithreaded programming code 300 may represent both the original program before it is ported to a new platform and the program after it is transferred to be executed on a new platform. A dynamic binary translator 303 may include a middle tier layer 304. An application programming interface (API) 306 may be used as an interface between the multithreaded programming code 300 and the dynamic binary translator 303.

[0034] An embodiment of the invention may define a minimum set of APIs in the API 306 to be used for communication between the multithreaded programming code 300 and the dynamic binary translator 303. For example, mie_init_translator( ) and mie_unload_translator( ) may be defined in the API 306 to initialize and release the dynamic binary translator 303. In addition, API calls such as mie_thread_init( ), mie_complete_thread_init( ), and mie_thread_term( ) may be defined in the API 306 to initialize new threads, ~~completing~~ complete the thread initialization and ~~terminating~~ terminate the threads. Thread creation and termination function calls 301, such as mie_thread_init( ), mie_complete_thread_init( ) and mie_thread_terms( ) may be called through the API 306 into the dynamic binary translator 303 and the middle tier layer 304.

[0040] A similar process is performed for the child thread 460 to associate the resources allocated [[or]] for the foreign thread and the child thread 460. As discussed above, the child thread 460 is created as the result of the CreateThread( ) 404. In an embodiment of the invention, the child thread 460 calls mie_complete_thread_init(CHILD) in operation 407. It is the same function call as the one with respect to associating the parent thread and the foreign thread resources. In this situation, a flag "CHILD" may be provided to indicate that the thread completion is to be performed in the context of the child thread 460. The operating system wrapper 440 may provide the starting point wherein the completion process may begin. As a result, the thread completion process associates the resources allocated for the foreign thread and the child thread 460.

[0042] In this example, the 64-bit multithread program calls an IPF operating system to create an IA-32 thread. Initially, the program calls mie_thread_init( ) to notify the middle tier layer 410. In mie_thread_init( ), resource is allocated for the IA-32 thread. Subsequently, the program sends an operating system request 403 to create the IA-32 thread. This request is intercepted by the operating system wrapper 440 and the mie_thread_init( ) call is returned to the middle tier layer 410 at this point. This point of suspension may be referred to as the clone point.

[0044] To associate the IA-32 thread and the IPF thread, the program calls mie_complete_thread_init(PARENT) 406. In this function, the parent IPF thread completes its

thread initialization by creating the IA-32 thread beginning from the clone point. Furthermore, the child IPF thread also ~~complete~~ completes its thread initialization by calling mie_complete_tread init(CHILD) 406. In this function, the IA-32 thread is translated and the resource context also begins from the clone point until the execution returns at the middle tier layer 410.